Python Code

Your Name

December 20, 2024

1 Task 1

Q1 : the minimum number of 3D points needed to sample in an iteration to compute a putative model.

A1 : the min number of 3D points we need to sample in an iteration to compute a putative model is 3 points. since we need 3 points to define a plan in the 3D space.

Q2 :Determine the probability that the data picked for to fit the putative model in a single iteration fails, assuming that the outlier ratio in the dataset is 0.5 and we are fitting 3D planes.

A2 : single iteration fails means there is at least one point in the 3 points we choose is outlier. so we can first calculate the probability that no outlier in the 3 points $--> 0.5^3 = 0.125$, then we know the probability that a single iteration fails is $1 - 0.125 = 0.875$

Q3 : Determine the minimum number of RANSAC trials needed to have $\geq 98\%$ chance of success, assuming that the outlier ratio in the dataset is 0.5 and we are fitting planes.

A3 : more than or equal to 98% success means less than or equal to 2% fails. according to the answer we got from question2, we know the single time fails is 0.875. so we can then get the equation $0.875^{trialsnum} \leq 0.02$ the trials number must bigger or equal to 29.3. round to the integer is 30.

2 Task 2

Q1 : the number of degrees of freedom M has and the minimum number of 2D correspondences that are required to fully constrain or estimate M.

A1 : the matrix M is 2*2 and has 4 independent elements. so the number of degrees of freedom would be 4. And, we need to have at least 2 correspondence 2D points to fully constrain M the reason is for each correspondent points, we can have 2 equations, so 2 correspondent points will then generate 4 equations in order to calculate the for elements in the matrix M.

Q2 :Write the form of A, m, and b.

A2 : the matrix M is $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ and we have the point $[x_i, y_i]$ corresponding to the point $[x_i', y_i']$ so we can then write two equations:
$ax_i + by_i = x_i'$
$cx_i + dy_i = y_i'$
then our next step is writing the equations to the matrix multiplication form:

$$\begin{bmatrix} x_i & y_i & 0 & 0 \\ 0 & 0 & x_i & y_i \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} x_i' \\ y_i' \end{bmatrix}$$

so if we have n data, then A is: $\begin{bmatrix} x_1 & y_1 & 0 & 0 \\ 0 & 0 & x_1 & y_1 \\ . & . & . & . \\ x_n & y_n & 0 & 0 \\ 0 & 0 & x_n & y_n \end{bmatrix}$ m is $\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$ b = $\begin{bmatrix} x_1' \\ y_1' \\ . \\ x_n' \\ y_n' \end{bmatrix}$

3 Task 3

Q1 :Report (S,t) in your report for points_case_1.npy.

A1 :compared with the task2, task three has a new term t, so the new expression should like:

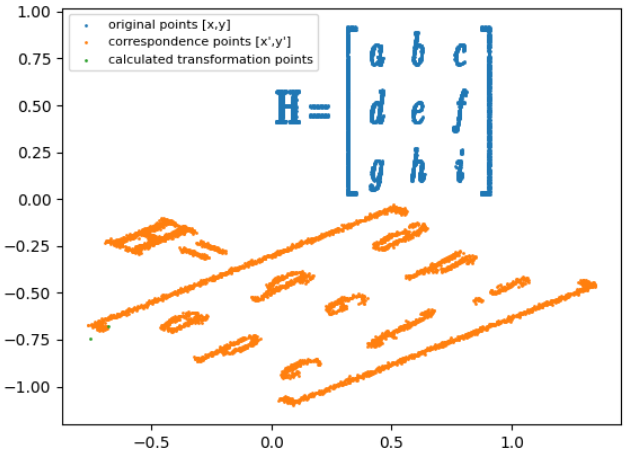
$$\begin{bmatrix} x_i & y_i & 0 & 0 & 1 & 0 \\ 0 & 0 & x_i & y_i & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix} = \begin{bmatrix} x_i' \\ y_i' \end{bmatrix}$$ and the parameter e and f come from the the t = $\begin{bmatrix} e \\ f \end{bmatrix}$

S is = $\begin{bmatrix} 1.41 & -1.41 \\ -0.71 & -0.71 \end{bmatrix}$ and the t is = $\begin{bmatrix} 0.10 \\ 0.20 \end{bmatrix}$

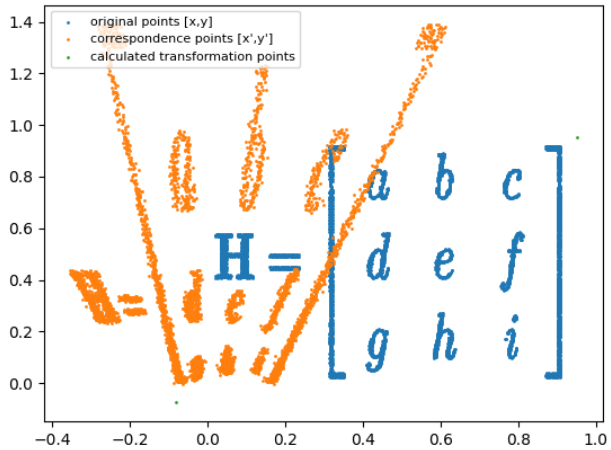
Q2 : Save the figures and put them in your report

A2 :

case_1:



case_2:



Q3 :how well does an affine transform describe the relationship for points_case_1 and points_case_2

A3 : the S and t parameter we calculated is really accurate and can transform the image really well (from the plot we can see the given correspondence points are almost same as the points we calculated).
for case_1: all the lines which are parallel before the transformation are still parallel after the transmission.

for case_2: the horizontal lines which are parallel before are still parallel after transmission, however form the plot when can see the vertical line which are parallel before are no longer parallel after transmission. they will cross at the same vanish point.

4 Task 4

Q1 : Fill in fit_homography in homography.py

A1 : the A will be $\begin{bmatrix} -x_i & -y_i & -1 & 0 & 0 & 0 & x_i x_i' & y_i x_i' & x_i' \\ 0 & 0 & 0-x_i & -y_i & -1 & & x_i y_i' & y_i y_i' & y_i' \end{bmatrix}$

and the h will be $\begin{bmatrix} h1 \\ h2 \\ h3 \\ h4 \\ h5 \\ h6 \\ h7 \\ h8 \\ h9 \end{bmatrix}$

what we need to do is to solve the equation: $\begin{bmatrix} -x_i & -y_i & -1 & 0 & 0 & 0 & x_i x_i' & y_i x_i' & x_i' \\ 0 & 0 & 0-x_i & -y_i & -1 & & x_i y_i' & y_i y_i' & y_i' \end{bmatrix} \begin{bmatrix} h1 \\ h2 \\ h3 \\ h4 \\ h5 \\ h6 \\ h7 \\ h8 \\ h9 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

if we have n points, the general equation would be:

$$\begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1 x_1' & y_1 x_1' & x_1' \\ 0 & 0 & 0-x_1 & -y_1 & -1 & & x_1 y_1' & y_1 y_1' & y_1' \\ . & . & . & . & . & . & . & . & . \\ -x_n & -y_n & -1 & 0 & 0 & 0 & x_n x_n' & y_n x_n' & x_n' \\ 0 & 0 & 0-x_n & -y_n & -1 & & x_n y_n' & y_n y_n' & y_n' \end{bmatrix} \begin{bmatrix} h1 \\ h2 \\ h3 \\ h4 \\ h5 \\ h6 \\ h7 \\ h8 \\ h9 \end{bmatrix} = \begin{bmatrix} 0 \\ . \\ 0 \end{bmatrix}$$

```
1
2  def fit_homography(XY):
3
4      '''
5      Given a set of N correspondences XY of the form [x,y,x',y'],
6      fit a homography from [x,y,1] to [x',y',1].
7
8      Input - XY: an array with size(N,4), each row contains two
9              points in the form [x_i, y_i, x'_i, y'_i] (1,4)
10     Output -H: a (3,3) homography matrix that (if the correspondences can be
11             described by a homography) satisfies [x',y',1]^T === H [x,y,1]^T
12
13     '''
14
15
16
17
18     datamatrix = XY
19     original_matrix = datamatrix[:,:2]
20     correspondece_matrix = datamatrix[:,2:]
21
22
23
24
25     x = original_matrix[:,0]
```

```
    y = original_matrix[:,1]



    x_c = correspondece_matrix[:,0]
    y_c = correspondece_matrix[:,1]

    H = datamatrix.shape[0]
    print(H)

    A = np.zeros((H*2,9))
    for i in range(H):
        A[2*i,0] = -x[i]
        A[2*i,1] = -y[i]
        A[2*i,2] = -1
        A[2*i,6] = x[i]*x_c[i]
        A[2*i,7] = y[i]*x_c[i]
        A[2*i,8] = x_c[i]

        A[2*i+1,3] = -x[i]
        A[2*i+1,4] = -y[i]
        A[2*i+1,5] = -1
        A[2*i+1,6] = x[i]*y_c[i]
        A[2*i+1,7] = y[i]*y_c[i]
        A[2*i+1,8] = y_c[i]


    # Compute the SVD of A
    _, _, V = np.linalg.svd(A)

    # The solution h is the last column of V
    h = V[-1]
    # Reshape h into a 3x3 matrix
     H_matrix = h.reshape((3, 3))
    print(H_matrix)
    return 0
```

Q2 :Report H for cases points_case_1.npy and points_case_4.npy. You must normalize the last entry to 1.

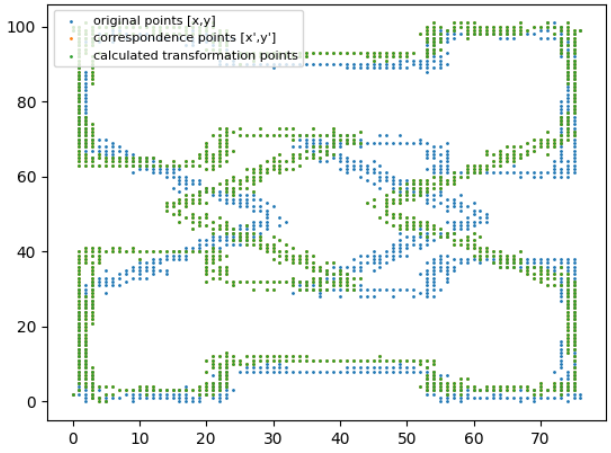
A2 : points_case_1: $\begin{bmatrix} 1.006 & 0.001613 & -0.135 \\ 0.00256 & 0.623 & -0.736 \\ 0 & 0 & 1 \end{bmatrix}$

points_case_4: $\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

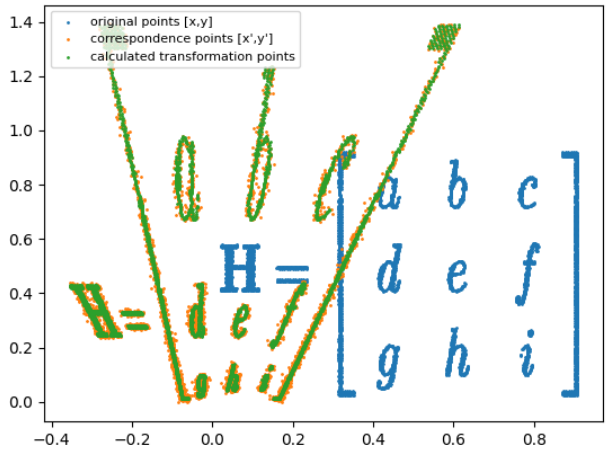
Q3 :Save the figure and put it in the report.

A3 :

the plot from points_case_5:



the plot from points_case_9:



5 Task 5

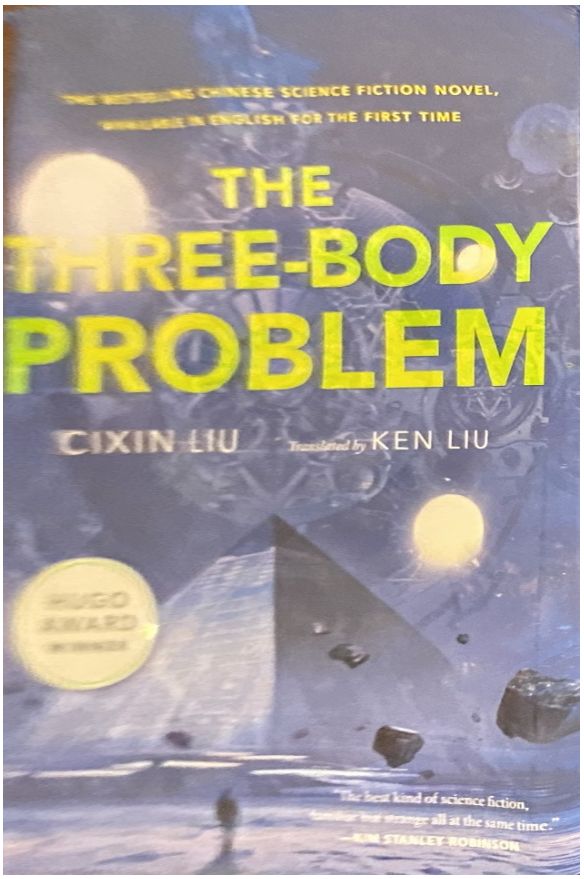
Q1 : Fill in make_synthetic_view(sceneImage,corners,size)

A1 :

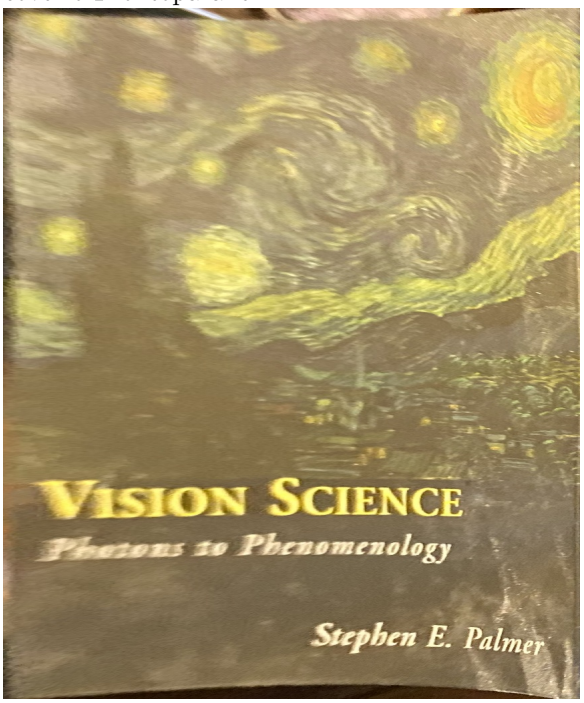
```
    def make_synthetic_view(img, corners, size):
    '''
    Creates an image with a synthetic view of selected region in the image
    from the front. The region is bounded by a quadrilateral denoted by the
    corners array. The size array defines the size of the final image.

    Input - img: image file of shape (H,W,3)
            corner: array containing corners of the book cover in
            the order [top-left, top-right, bottom-right, bottom-left]  (4,2)
            size: array containing size of book cover in inches [height, width] (1,2)

    Output - A fronto-parallel view of selected pixels (the book as if the cover is
            parallel to the image plane), using 100 pixels per inch.
    '''
    h,w = size[0] * 100


    new_corners =  np.array([[0, 0], [w - 1, 0], [w - 1, h - 1], [0, h - 1]], dtype=np.float32)
    M = cv2.getPerspectiveTransform(corners.astype(np.float32), new_corners)
    synthetic_view = cv2.warpPerspective(img, M, (int(w), int(h)))
    return synthetic_view


```

Q2 : Put a copy of both book covers in your report.

A2 :
cover of threebogy:



cover of_frontoparallel



Q3 : why you think the lines might be slightly crooked despite the book cover being roughly a plane.

A3 :After observing two images, i thing the cover of book palmer doesn't have perfectly straight lines. I think the reason is, when we use the cv.warpperspective, we only use four points to describe the original corner (position) of the book,

which means we assume the edges of the book are perfectly straight. However, from the original image of the book palmer, we can see the edge of the book is curve and the affine transformation we use here is linear transformation, so it can't transform the straight line to curve or curve to straight line.

6 Task 6

Q1 : Fill in compute_distance in task6.py

A1 :

```
    def compute_distance(desc1, desc2):
    '''
    Calculates L2 distance between 2 binary descriptor vectors.

    Input - desc1: Descriptor vector of shape (N,F)
            desc2: Descriptor vector of shape (M,F)

    Output - dist: a (N,M) L2 distance matrix where dist(i,j)
             is the squared Euclidean distance between row i of
             desc1 and desc2. You may want to use the distance
             calculation trick
             ||x - y||^2 = ||x||^2 + ||y||^2 - 2x^T y
    '''
    norm_desc_1 = np.linalg.norm(desc1, axis=1, keepdims=True)
    norm_desc_2 = np.linalg.norm(desc2, axis=1, keepdims=True)

    distant = norm_desc_1 ** 2 + (norm_desc_2 ** 2).T - 2 * np.dot(desc1, desc2.T)

    return distant

```

Q2 :Fill in find_matches in task6.py

A2 :

```
        def find_matches(desc1, desc2, ratioThreshold):
    '''
    Calculates the matches between the two sets of keypoint
    descriptors based on distance and ratio test.

    Input - desc1: Descriptor vector of shape (N,F)
            desc2: Descriptor vector of shape (M,F)
            ratioThreshhold : maximum acceptable distance ratio between 2
                              nearest matches

    Output - matches: a list of indices (i,j) 1 <= i <= N, 1 <= j <= M giving
             the matches between desc1 and desc2.

             This should be of size (K,2) where K is the number of
             matches and the row [ii,jj] should appear if desc1[ii,:] and
             desc2[jj,:] match.
    '''
    matches = []

    # Calculate distances between descriptors
    dist = compute_distance(desc1, desc2)

    # Find the nearest neighbor for each descriptor in desc1
    nearest_neighbor_indices = np.argmin(dist, axis=1)

    # Compute distance ratios
    min_dist = dist[np.arange(len(desc1)), nearest_neighbor_indices]
    sec_nearest_neighbor_dist = np.partition(dist, 1, axis=1)[:, 1]
    dist_ratio = min_dist / sec_nearest_neighbor_dist

    # Apply ratio test and add matches to the list
    for i, ratio in enumerate(dist_ratio):
        if ratio < ratioThreshold:
            matches.append([i, nearest_neighbor_indices[i]])

    return np.array(matches)

```

```

```

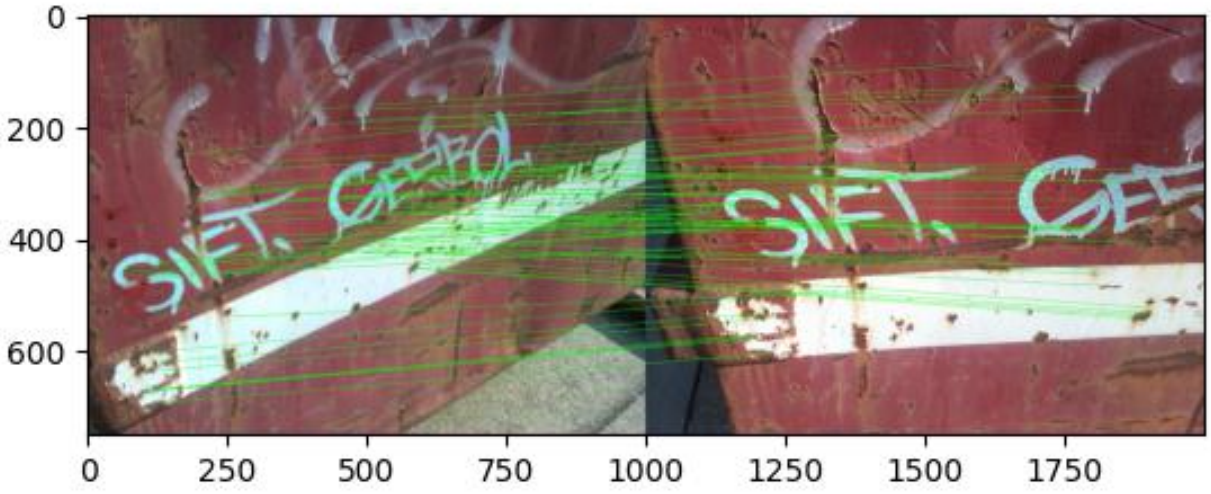
Q3 :Fill in draw_matches

A3 :

```
        def draw_matches(img1, img2, kp1, kp2, matches):
    '''
    Creates an output image where the two source images stacked vertically
    connecting matching keypoints with a line.

    Input - img1: Input image 1 of shape (H1,W1,3)
            img2: Input image 2 of shape (H2,W2,3)
            kp1: Keypoint matrix for image 1 of shape (N,4)
            kp2: Keypoint matrix for image 2 of shape (M,4)
            matches: List of matching pairs indices between the 2 sets of
                     keypoints (K,2)

    Output - Image where 2 input images stacked vertically with lines joining
             the matched keypoints
    Hint: see cv2.line
    '''
    #Hint:
    #Use common.get_match_points() to extract keypoint locations

    h1, w1 = img1.shape[:2]
    h2, w2 = img2.shape[:2]

    output_height = max(h1, h2)
    output_width = w1 + w2

    output = np.zeros((output_height, output_width, 3), dtype=np.uint8)
    output[:h1, :w1] = img1
    output[:h2, w1:] = img2

    matched_rows = common.get_match_points(kp1, kp2, matches)
    for x1, y1, x2, y2 in matched_rows:
        # Shift the x-coordinate of the second point to account for the width of the first image
        x2 += w1
        cv2.line(output, (int(x1), int(y1)), (int(x2), int(y2)), (0, 255, 0), 1)


    return output

```

Q4 : Put a picture of the matches between two image pairs of your choice in your report.

A4 :



Q5 :Fill in RANSAC_fit_homography

A5 :

```
    def RANSAC_fit_homography(XY, eps=1, nIters=1000):
'''
Perform RANSAC to find the homography transformation
matrix which has the most inliers

Input - XY: an array with size(N,4), each row contains two
        points in the form [x_i, y_i, x'_i, y'_i] (1,4)
        eps: threshold distance for inlier calculation
        nIters: number of iteration for running RANSAC
Output - bestH: a (3,3) homography matrix fit to the
                inliers from the best model.

Hints:
a) Sample without replacement. Otherwise you risk picking a set of points
   that have a duplicate.
b) *Re-fit* the homography after you have found the best inliers
'''
bestH, bestCount, bestInliers = np.eye(3), -1, np.zeros((XY.shape[0],))

for _ in range(nIters):
    # Randomly sample 4 points without replacement
    idx = np.random.choice(XY.shape[0], 4, replace=False)
    sample = XY[idx]

    # Fit homography to the sampled points
    H = fit_homography(sample)

    # Transform all points using the homography
```

```
        originalXY = XY[:, :2]
        transformedXY = homography_transform(originalXY, H)

        # Calculate distances between transformed and actual points
        distances = np.linalg.norm(XY[:, 2:] - transformedXY, axis=1)

        # Count inliers based on distance threshold
        inliers = distances < eps
        inlier_count = np.sum(inliers)

        # Update best model if current model has more inliers
        if inlier_count > bestCount:
            bestCount = inlier_count
            bestInliers = XY[inliers]
            bestH = fit_homography(bestInliers)

    return bestH
```

Q6 : Fill in make_warped and warp_and_combine

A6 :

```
        def make_warped(img1, img2):
    '''
    Take two images and return an image, putting together the full pipeline.
    You should return an image of the panorama put together.

    Input - img1: Input image 1 of shape (H1,W1,3)
            img2: Input image 1 of shape (H2,W2,3)

    Output - Final stitched image
    Be careful about:
    a) The final image size
    b) Writing code so that you first estimate H and then merge images with H.
    The system can fail to work due to either failing to find the homography or
    failing to merge things correctly.
    '''
    kp1, desc1= common.get_AKAZE(img1)
    kp2, desc2 = common.get_AKAZE(img2)
    matches = find_matches(desc1, desc2, ratioThreshold = 0.7)
    XY = common.get_match_points(kp1, kp2, matches)

    H = RANSAC_fit_homography(XY)

    stitched = warp_and_combine(img2, img1, H)
    return stitched
```

```
        def warp_and_combine(img1, img2, H):
    '''
    You may want to write a function that merges the two images together given
    the two images and a homography: once you have the homography you do not
    need the correspondences; you just need the homography.
    Writing a function like this is entirely optional, but may reduce the chance
    of having a bug where your homography estimation and warping code have odd
    interactions.

    Input - img1: Input image 1 of shape (H1,W1,3)
            img2: Input image 2 of shape (H2,W2,3)
            H: homography mapping betwen them
    Output - V: stitched image of size (?,?,3); unknown since it depends on H
    '''
    # Get the dimensions of the input images
    H1, W1 = img1.shape[:2]
    H2, W2 = img2.shape[:2]

    # Define the corners of the input images
    corners1 = np.array([[0, 0], [W1-1, 0], [W1-1, H1-1], [0, H1-1]])
    corners2 = np.array([[0, 0], [W2-1, 0], [W2-1, H2-1], [0, H2-1]])

    # Transform corners of img1 to img2's coordinate system using the homography
    corners2_transformed = cv2.perspectiveTransform(corners1.reshape(-1, 1, 2), H).reshape(-1, 2)

    # Calculate the bounding box of the combined image
```

```
    xmin = int(min(0, np.min(corners2_transformed[:, 0])), 0))
    ymin = int(min(0, np.min(corners2_transformed[:, 1])), 0))
    xmax = int(max(W2 - 1, np.max(corners2_transformed[:, 0]), W1 - 1))
    ymax = int(max(H2 - 1, np.max(corners2_transformed[:, 1]), H1 - 1))

    # Calculate the size of the combined image
    width = xmax - xmin + 1
    height = ymax - ymin + 1

    # Calculate the translation matrix to shift the images
    T = np.array([[1, 0, -xmin], [0, 1, -ymin], [0, 0, 1]])

    # Warp img1 and img2 to the combined image coordinate system
    img1_warped = cv2.warpPerspective(img1, T.dot(H), (width, height))
    img2_warped = cv2.warpPerspective(img2, T, (width, height))

    # Combine the warped images
    V = cv2.addWeighted(img1_warped, 0.5, img2_warped, 0.5, 0)

    return V

```

Q7 :Put merges from two of your favorite pairs in the report.

A7 :

7 Task 7

Q1 :Fill in the function improve_image(scene,template,transfer)

A1 :

```
            def improve_image(scene, template, transfer):
    '''
    Detect template image in the scene image and replace it with transfer image.

    Input - scene: image (H,W,3)
            template: image (K,K,3)
            transfer: image (L,L,3)
    Output - augment: the image with

    Hints:
    a) You may assume that the template and transfer are both squares.
    b) This will work better if you find a nearest neighbor for every template
       keypoint as opposed to the opposite, but be careful about directions of the
       estimated homography and warping!
    '''
     # Extract keypoints and descriptors from scene and template images
    kps1, d1 = common.get_AKAZE(scene)
    kps2, d2 = common.get_AKAZE(template)

    # Find matches between descriptors of scene and template images
    matches = find_matches(d1, d2, 0.7)

    # Get corresponding keypoints coordinates
    XY = common.get_match_points(kps1, kps2, matches)

    # Calculate homography matrix using RANSAC
    H = RANSAC_fit_homography(XY)

    # Scale the template to the size of the transfer image
    scaled_template = cv2.resize(template, (transfer.shape[1], transfer.shape[0]))

    # Calculate transformation matrix to warp transfer image to align with template image
    Rtrans2temp = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
    Rtemp2trans = np.array([[transfer.shape[1] / template.shape[1], 0, 0], [0, transfer.shape
    [0] / template.shape[0], 0], [0, 0, 1]])
    transform_matrix = np.dot(np.dot(np.linalg.inv(H), Rtrans2temp), Rtemp2trans)

    # Warp the transfer image to align with the template image
    warped_transfer = cv2.warpPerspective(transfer, transform_matrix, (scene.shape[1], scene.
    shape[0]))

    # Combine the scene image with the warped transfer image
    augment = task7_warp_and_combine(scene, warped_transfer)

    return augment

```

Q2 :Include the images in your report.

A2 :


NATURALLY
PAMPLEMOUSSE
SPARKLING WATER
ESSENCED
LaCroix